

# CHAPTER 4

## SOFTWARE CONSTRUCTION

### ACRONYMS

OMG	Object Management Group
UML	Unified Modeling Language

### INTRODUCTION

The term software construction refers to the detailed creation of working, meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging.

The Software Construction Knowledge Area is linked to all the other KAs, most strongly to Software Design and Software Testing. This is because the software construction process itself involves significant software design and test activity. It also uses the output of design and provides one of the inputs to testing, both design and testing being the activities, not the KAs in this case. Detailed boundaries between design, construction, and testing (if any) will vary depending upon the software life cycle processes that are used in a project.

Although some detailed design may be performed prior to construction, much design work is performed within the construction activity itself. Thus the Software Construction KA is closely linked to the Software Design KA.

Throughout construction, software engineers both unit-test and integration-test their work. Thus, the Software Construction KA is closely linked to the Software Testing KA as well.

Software construction typically produces the highest volume of configuration items that need to be managed in a software project (source files, content, test cases, and so on). Thus, the Software Construction KA is also closely linked to the Software Configuration Management KA.

Since software construction relies heavily on tools and methods and is probably the most tool-intensive of the KAs, it is linked to the Software Engineering Tools and Methods KA.

While software quality is important in all the KAs, code is the ultimate deliverable of a software project, and thus Software Quality is also closely linked to Software Construction.

Among the Related Disciplines of Software Engineering, the Software Construction KA is most akin to computer science in its use of knowledge of algorithms and of detailed coding practices, both of which are often considered

to belong to the computer science domain. It is also related to project management, insofar as the management of construction can present considerable challenges.

### BREAKDOWN OF TOPICS FOR SOFTWARE CONSTRUCTION

The breakdown of the Software Construction KA is presented below, together with brief descriptions of the major topics associated with it. Appropriate references are also given for each of the topics. Figure 1 gives a graphical representation of the top-level decomposition of the breakdown for this KA.

#### 1. Software Construction Fundamentals

The fundamentals of software construction include

- ♦ Minimizing complexity
- ♦ Anticipating change
- ♦ Constructing for verification
- ♦ Standards in construction

The first three concepts apply to design as well as to construction. The following sections define these concepts and describe how they apply to construction.

##### 1.1. *Minimizing Complexity*

[Bec99; Ben00; Hun00; Ker99; Mag93; McC04]

A major factor in how people convey intent to computers is the severely limited ability of people to hold complex structures and information in their working memories, especially over long periods of time. This leads to one of the strongest drivers in software construction: minimizing complexity. The need to reduce complexity applies to essentially every aspect of software construction, and is particularly critical to the process of verification and testing of software constructions.

In software construction, reduced complexity is achieved through emphasizing the creation of code that is simple and readable rather than clever.

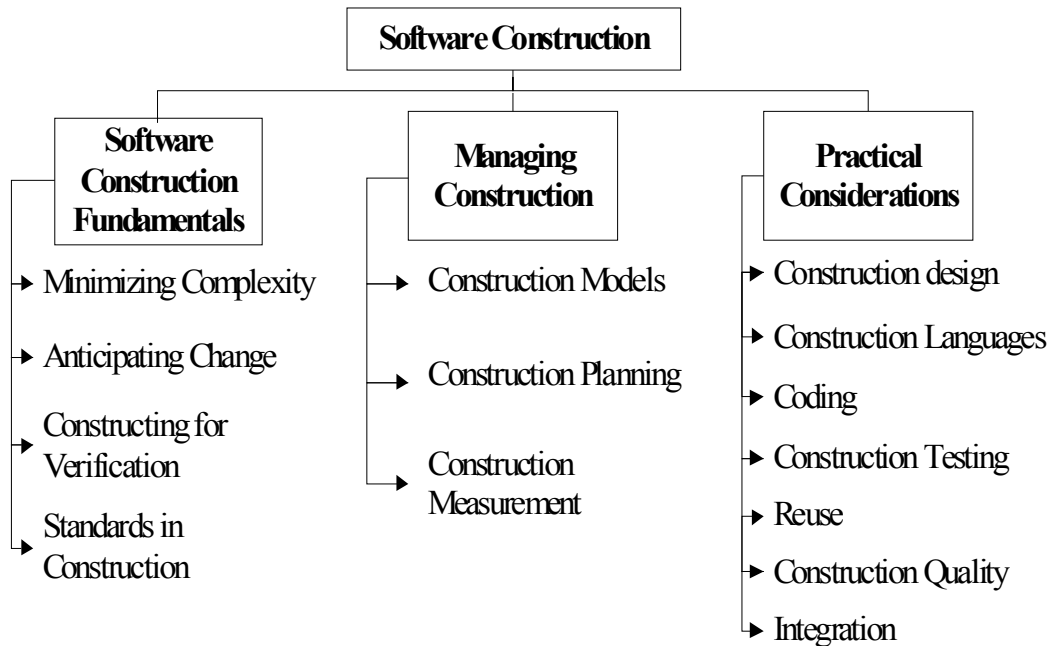
Minimizing complexity is accomplished through making use of standards, which is discussed in topic 1.4 *Standards in Construction*, and through numerous specific techniques which are summarized in topic 3.3 *Coding*. It is also supported by the construction-focused quality techniques summarized in topic 3.5 *Construction Quality*.

1.2. *Anticipating Change*  
 [Ben00; Ker99; McC04]

Most software will change over time, and the anticipation of change drives many aspects of software construction. Software is unavoidably part of changing external environments, and changes in those outside environments affect software in diverse ways.

Anticipating change is supported by many specific techniques summarized in topic 3.3 *Coding*.

- ◆ Communication methods (for example, standards for document formats and contents)
- ◆ Programming languages (for example, language standards for languages like Java and C++)
- ◆ Platforms (for example, programmer interface standards for operating system calls)
- ◆ Tools (for example, diagrammatic standards for notations like UML (Unified Modeling Language))



**Figure 1.** Breakdown of topics for the Software Construction KA.

1.3. *Constructing for Verification*  
 [Ben00; Hun00; Ker99; Mag93; McC04]

Constructing for verification means building software in such a way that faults can be ferreted out readily by the software engineers writing the software, as well as during independent testing and operational activities. Specific techniques that support constructing for verification include following coding standards to support code reviews, unit testing, organizing code to support automated testing, and restricted use of complex or hard-to-understand language structures, among others.

1.4. *Standards in Construction*  
 [IEEE12207-95; McC04]

Standards that directly affect construction issues include

*Use of external standards.* Construction depends on the use of external standards for construction languages, construction tools, technical interfaces, and interactions between Software Construction and other KAs. Standards come from numerous sources, including hardware and software interface specifications such as the Object Management Group (OMG) and international organizations such as the IEEE or ISO.

*Use of internal standards.* Standards may also be created on an organizational basis at the corporate level or for use on specific projects. These standards support coordination of group activities, minimizing complexity, anticipating change, and constructing for verification.

## 2. Managing Construction

### 2.1. Construction Models [Bec99; McC04]

Numerous models have been created to develop software, some of which emphasize construction more than others.

Some models are more linear from the construction point of view, such as the waterfall and staged-delivery life cycle models. These models treat construction as an activity which occurs only after significant prerequisite work has been completed—including detailed requirements work, extensive design work, and detailed planning. The more linear approaches tend to emphasize the activities that precede construction (requirements and design), and tend to create more distinct separations between the activities. In these models, the main emphasis of construction may be coding.

Other models are more iterative, such as evolutionary prototyping, Extreme Programming, and Scrum. These approaches tend to treat construction as an activity that occurs concurrently with other software development activities, including requirements, design, and planning, or overlaps them. These approaches tend to mix design, coding, and testing activities, and they often treat the combination of activities as construction.

Consequently, what is considered to be “construction” depends to some degree on the life cycle model used.

### 2.2. Construction Planning [Bec99; McC04]

The choice of construction method is a key aspect of the construction planning activity. The choice of construction method affects the extent to which construction prerequisites are performed, the order in which they are performed, and the degree to which they are expected to be completed before construction work begins.

The approach to construction affects the project’s ability to reduce complexity, anticipate change, and construct for verification. Each of these objectives may also be addressed at the process, requirements, and design levels—but they will also be influenced by the choice of construction method.

Construction planning also defines the order in which components are created and integrated, the software quality management processes, the allocation of task assignments to specific software engineers, and the other tasks, according to the chosen method.

### 2.3. Construction Measurement [McC04]

Numerous construction activities and artifacts can be measured, including code developed, code modified, code reused, code destroyed, code complexity, code inspection statistics, fault-fix and fault-find rates, effort, and scheduling. These measurements can be useful for purposes of managing construction, ensuring quality during construction,

improving the construction process, as well as for other reasons. See the Software Engineering Process KA for more on measurements.

## 3. Practical considerations

Construction is an activity in which the software has to come to terms with arbitrary and chaotic real-world constraints, and to do so exactly. Due to its proximity to real-world constraints, construction is more driven by practical considerations than some other KAs, and software engineering is perhaps most craft-like in the construction area.

### 3.1. Construction Design [Bec99; Ben00; Hun00; IEEE12207-95; Mag93; McC04]

Some projects allocate more design activity to construction; others to a phase explicitly focused on design. Regardless of the exact allocation, some detailed design work will occur at the construction level, and that design work tends to be dictated by immovable constraints imposed by the real-world problem that is being addressed by the software.

Just as construction workers building a physical structure must make small-scale modifications to account for unanticipated gaps in the builder’s plans, software construction workers must make modifications on a smaller or larger scale to flesh out details of the software design during construction.

The details of the design activity at the construction level are essentially the same as described in the Software Design KA, but they are applied on a smaller scale.

### 3.2. Construction Languages [Hun00; McC04]

*Construction languages* include all forms of communication by which a human can specify an executable problem solution to a computer.

The simplest type of construction language is a *configuration language*, in which software engineers choose from a limited set of predefined options to create new or custom software installations. The text-based configuration files used in both the Windows and Unix operating systems are examples of this, and the menu style selection lists of some program generators constitute another.

*Toolkit languages* are used to build applications out of toolkits (integrated sets of application-specific reusable parts), and are more complex than configuration languages. Toolkit languages may be explicitly defined as application programming languages (for example, scripts), or may simply be implied by the set of interfaces of a toolkit.

*Programming languages* are the most flexible type of construction languages. They also contain the least amount of information about specific application areas and

development processes, and so require the most training and skill to use effectively.

There are three general kinds of notation used for programming languages, namely:

- ♦ Linguistic
- ♦ Formal
- ♦ Visual

*Linguistic notations* are distinguished in particular by the use of word-like strings of text to represent complex software constructions, and the combination of such word-like strings into patterns that have a sentence-like syntax. Properly used, each such string should have a strong semantic connotation providing an immediate intuitive understanding of what will happen when the underlying software construction is executed.

*Formal notations* rely less on intuitive, everyday meanings of words and text strings and more on definitions backed up by precise, unambiguous, and formal (or mathematical) definitions. Formal construction notations and formal methods are at the heart of most forms of system programming, where accuracy, time behavior, and testability are more important than ease of mapping into natural language. Formal constructions also use precisely defined ways of combining symbols that avoid the ambiguity of many natural language constructions.

*Visual notations* rely much less on the text-oriented notations of both linguistic and formal construction, and instead rely on direct visual interpretation and placement of visual entities that represent the underlying software. Visual construction tends to be somewhat limited by the difficulty of making “complex” statements using only movement of visual entities on a display. However, it can also be a powerful tool in cases where the primary programming task is simply to build and “adjust” a visual interface to a program, the detailed behavior of which has been defined earlier.

### 3.2. Coding

[Ben00; IEEE12207-95; McC04]

The following considerations apply to the software construction coding activity:

- ♦ Techniques for creating understandable source code, including naming and source code layout
- ♦ Use of classes, enumerated types, variables, named constants, and other similar entities
- ♦ Use of control structures
- ♦ Handling of error conditions—both planned errors and exceptions (input of bad data, for example)
- ♦ Prevention of code-level security breaches (buffer overruns or array index overflows, for example)
- ♦ Resource usage via use of exclusion mechanisms and discipline in accessing serially reusable resources (including threads or database locks)

- ♦ Source code organization (into statements, routines, classes, packages, or other structures)
- ♦ Code documentation
- ♦ Code tuning

### 3.3. Construction Testing

[Bec99; Hun00; Mag93; McC04]

Construction involves two forms of testing, which are often performed by the software engineer who wrote the code:

- ♦ Unit testing
- ♦ Integration testing

The purpose of construction testing is to reduce the gap between the time at which faults are inserted into the code and the time those faults are detected. In some cases, construction testing is performed after code has been written. In other cases, test cases may be created before code is written.

Construction testing typically involves a subset of types of testing, which are described in the Software Testing KA. For instance, construction testing does not typically include system testing, alpha testing, beta testing, stress testing, configuration testing, usability testing, or other, more specialized kinds of testing.

Two standards have been published on the topic: IEEE Std 829-1998, *IEEE Standard for Software Test Documentation* and IEEE Std 1008-1987, *IEEE Standard for Software Unit Testing*.

See also the corresponding sub-topics in the Software Testing KA: 2.1.1 *Unit Testing* and 2.1.2 *Integration Testing* for more specialized reference material.

### 3.4. Reuse

[IEEE1517-99; Som05].

As stated in the introduction of (IEEE1517-99):

“Implementing software reuse entails more than creating and using libraries of assets. It requires formalizing the practice of reuse by integrating reuse processes and activities into the software life cycle.” However, reuse is important enough in software construction that it is included here as a topic.

The tasks related to reuse in software construction during coding and testing are:

- ♦ The selection of the reusable units, databases, test procedures, or test data
- ♦ The evaluation of code or test reusability
- ♦ The reporting of reuse information on new code, test procedures, or test data

### 3.5. Construction Quality

[Bec99; Hun00; IEEE12207-95; Mag93; McC04]

Numerous techniques exist to ensure the quality of code as it is constructed. The primary techniques used for construction include

- ◆ Unit testing and integration testing (as mentioned in topic 3.4 *Construction Testing*)
- ◆ Test-first development (see also the Software Testing KA, topic 2.2 *Objectives of Testing*)
- ◆ Code stepping
- ◆ Use of assertions
- ◆ Debugging
- ◆ Technical reviews (see also the Software Quality KA, sub-topic 2.3.2 *Technical Reviews*)
- ◆ Static analysis (IEEE1028) (see also the Software Quality KA, topic 2.3 *Reviews and Audits*)

The specific technique or techniques selected depend on the nature of the software being constructed, as well as on the skills set of the software engineers performing the construction.

Construction quality activities are differentiated from other quality activities by their focus. Construction quality activities focus on code and on artifacts that are closely related to code: small-scale designs—as opposed to other artifacts that are less directly connected to the code, such as requirements, high-level designs, and plans.

### 3.7 *Integration*

[Bec99; IEEE12207-95; McC04]

A key activity during construction is the integration of separately constructed routines, classes, components, and subsystems. In addition, a particular software system may need to be integrated with other software or hardware systems.

Concerns related to construction integration include planning the sequence in which components will be integrated, creating scaffolding to support interim versions of the software, determining the degree of testing and quality work performed on components before they are integrated, and determining points in the project at which interim versions of the software are tested.

## Matrix of Topics vs. Reference Material

	[Bec99]	[Ben00]	[Hun00]	[IEEE 1517]	[IEEE 12207.0]	[Ker99]	[Mag93]	[Mcc04]	[Som05]
<b>1. Software Construction Fundamentals</b>									
<i>1.1 Minimizing Complexity</i>	c17	c2, c3	c7, c8			c2, c3	c6	c2, c3, c7-c9, c24, c27, c28, c31, c32, c34	
<i>1.2 Anticipating Change</i>		c11, c13, c14				c2, c9		c3-c5, c24, c31, c32, c34	
<i>1.3 Constructing for Verification</i>		c4	c21, c23, c34, c43			c1, c5, c6	c2, c3, c5, c7	c8, c20-c23, c31, c34	
<i>1.4 Standards in Construction</i>					X			c4	
<b>2. Managing Construction</b>									
<i>2.1 Construction Modals</i>	c10							c2, c3, c27, c29	
<i>2.2 Construction Planning</i>	c12, c15, c21							c3, c4, c21, c27-c29	
<i>2.3 Construction Measurement</i>								c25, c28	
<b>3. Practical Considerations</b>									
<i>3.1 Construction Design</i>	c17	c8-c10, p175-6	c33		X		c6	c3, c5, c24	
<i>3.2 Construction Languages</i>			c12, c14-c20					c4	
<i>3.3 Coding</i>		c6-c10			X			c5-c19, c25-c26	
<i>3.4 Construction Testing</i>	c18		c34, c43		X		c4	c22, c23	
<i>3.5 Reuse</i>				X					c14
<i>3.6 Construction Quality</i>	c18		c18		X		c4, c6, c7	c8, c20-c25	
<i>3.7 Integration</i>	c16				X			c29	

## RECOMMENDED REFERENCES FOR SOFTWARE CONSTRUCTION

- [Bec99] K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999, Chap. 10, 12, 15, 16-18, 21.
- [Ben00a] J. Bentley, *Programming Pearls*, second ed., Addison-Wesley, 2000, Chap. 2-4, 6-11, 13, 14, pp. 175-176.
- [Hun00] A. Hunt and D. Thomas, *The Pragmatic Programmer*, Addison-Wesley, 2000, Chap. 7, 8, 12, 14-21, 23, 33, 34, 36-40, 42, 43.
- [IEEE1517-99] IEEE Std 1517-1999, *IEEE Standard for Information Technology-Software Life Cycle Processes-Reuse Processes*, IEEE, 1999.
- [IEEE12207.0-96] IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, IEEE, 1996.
- [Ker99a] B.W. Kernighan and R. Pike, *The Practice of Programming*, Addison-Wesley, 1999, Chap. 2, 3, 5, 6, 9.
- [Mag93] S. Maguire, *Writing Solid Code: Microsoft's Techniques for Developing Bug-Free C Software*, Microsoft Press, 1993, Chap. 2-7.
- [McC04] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*, Microsoft Press, second ed., 2004.
- [Som05] I. Sommerville, *Software Engineering*, seventh ed., Addison-Wesley, 2005.

## APPENDIX A. LIST OF FURTHER READINGS

- (Bar98) T.T. Barker, *Writing Software Documentation: A Task-Oriented Approach*, Allyn & Bacon, 1998.
- (Bec02) K. Beck, *Test-Driven Development: By Example*, Addison-Wesley, 2002.
- (Fow99) M. Fowler and al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
- (How02) M. Howard and D.C. Leblanc, *Writing Secure Code*, Microsoft Press, 2002.
- (Hum97b) W.S. Humphrey, *Introduction to the Personal Software Process*, Addison-Wesley, 1997.
- (Mey97) B. Meyer, *Object-Oriented Software Construction*, second ed., Prentice Hall, 1997, Chap. 6, 10, 11.
- (Set96) R. Sethi, *Programming Languages: Concepts & Constructs*, second ed., Addison-Wesley, 1996, Parts II-V.

## **APPENDIX B. LIST OF STANDARDS**

(IEEE829-98) IEEE Std 829-1998, *IEEE Standard for Software Test Documentation*, IEEE, 1998.

(IEEE1008-87) IEEE Std 1008-1987 (R2003), *IEEE Standard for Software Unit Testing*, IEEE, 1987.

(IEEE1028-97) IEEE Std 1028-1997 (R2002), *IEEE Standard for Software Reviews*, IEEE, 1997.

(IEEE1517-99) IEEE Std 1517-1999, *IEEE Standard for Information Technology-Software Life Cycle Processes-Reuse Processes*, IEEE, 1999.

(IEEE12207.0-96) IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, IEEE, 1996.



# CHAPTER 5

## SOFTWARE TESTING

### ACRONYM

SRET	Software Reliability Engineered Testing
------	---

### INTRODUCTION

Testing is an activity performed for evaluating product quality, and for improving it, by identifying defects and problems.

Software testing consists of the *dynamic* verification of the behavior of a program on a *finite* set of test cases, suitably *selected* from the usually infinite executions domain, against the *expected* behavior.

In the above definition, italicized words correspond to key issues in identifying the Knowledge Area of Software Testing. In particular:

- ♦ *Dynamic*: This term means that testing always implies executing the program on (valued) inputs. To be precise, the input value alone is not always sufficient to determine a test, since a complex, nondeterministic system might react to the same input with different behaviors, depending on the system state. In this KA, though, the term “input” will be maintained, with the implied convention that its meaning also includes a specified input state, in those cases in which it is needed. Different from testing and complementary to it are static techniques, as described in the Software Quality KA.
- ♦ *Finite*: Even in simple programs, so many test cases are theoretically possible that exhaustive testing could require months or years to execute. This is why in practice the whole test set can generally be considered infinite. Testing always implies a trade-off between limited resources and schedules on the one hand and inherently unlimited test requirements on the other.
- ♦ *Selected*: The many proposed test techniques differ essentially in how they select the test set, and software engineers must be aware that different selection criteria may yield vastly different degrees of effectiveness. How to identify the most suitable selection criterion under given conditions is a very complex problem; in practice, risk analysis techniques and test engineering expertise are applied.
- ♦ *Expected*: It must be possible, although not always easy, to decide whether the observed outcomes of program execution are acceptable or not, otherwise the testing effort would be useless. The observed behavior may be checked against user expectations (commonly referred to as testing for validation), against a specification (testing for verification), or, finally, against the anticipated behavior from implicit

requirements or reasonable expectations. See, in the Software Requirements KA, topic 6.4 *Acceptance Tests*.

The view of software testing has evolved towards a more constructive one. Testing is no longer seen as an activity which starts only after the coding phase is complete, with the limited purpose of detecting failures. Software testing is now seen as an activity which should encompass the whole development and maintenance process and is itself an important part of the actual product construction. Indeed, planning for testing should start with the early stages of the requirement process, and test plans and procedures must be systematically and continuously developed, and possibly refined, as development proceeds. These test planning and designing activities themselves constitute useful input for designers in highlighting potential weaknesses (like design oversights or contradictions, and omissions or ambiguities in the documentation).

It is currently considered that the right attitude towards quality is one of prevention: it is obviously much better to avoid problems than to correct them. Testing must be seen, then, primarily as a means for checking not only whether the prevention has been effective, but also for identifying faults in those cases where, for some reason, it has not been effective. It is perhaps obvious but worth recognizing that, even after successful completion of an extensive testing campaign, the software could still contain faults. The remedy for software failures experienced after delivery is provided by corrective maintenance actions. Software maintenance topics are covered in the Software Maintenance KA.

In the Software Quality KA (See topic 3.3 *Software Quality Management Techniques*), software quality management techniques are notably categorized into *static* techniques (no code execution) and *dynamic* techniques (code execution). Both categories are useful. This KA focuses on dynamic techniques.

Software testing is also related to software construction (see topic 3.4 *Construction Testing* in that KA). Unit and integration testing are intimately related to software construction, if not part of it.

### BREAKDOWN OF TOPICS

The breakdown of topics for the Software Testing KA is shown in Figure 1.

The first subarea describes *Software Testing Fundamentals*. It covers the basic definitions in the field of software testing, the basic terminology and key issues, and its relationship with other activities.

The second subarea, *Test Levels*, consists of two (orthogonal) topics: 2.1 lists the levels in which the testing

of large software is traditionally subdivided; and 2.2 considers testing for specific conditions or properties and is referred to as *objectives of testing*. Not all types of testing apply to every software product, nor has every possible type been listed.

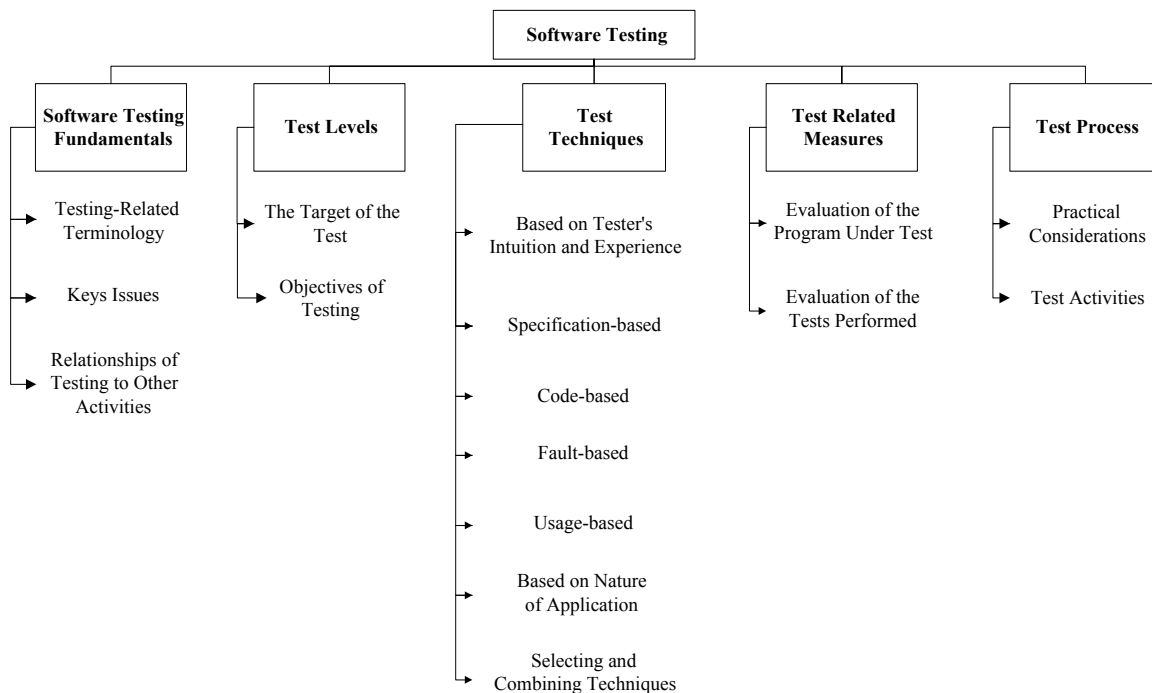
The test target and test objective together determine how the test set is identified, both with regard to its consistency—*how much testing is enough for achieving the stated objective*—and its composition—*which test cases should be selected for achieving the stated objective* (although usually the “for achieving the stated objective” part is left implicit and only the first part of the two

italicized questions above is posed). Criteria for addressing the first question are referred to as *test adequacy* criteria, while those addressing the second question are the *test selection* criteria.

Several *Test Techniques* have been developed in the past few decades, and new ones are still being proposed. Generally accepted techniques are covered in subarea 3.

*Test-related Measures* are dealt with in subarea 4.

Finally, issues relative to *Test Process* are covered in subarea 5.



**Figure 1** Breakdown of topics for the Software Testing KA

## 1. Software Testing Fundamentals

### 1.1. Testing-related terminology

1.1.1. Definitions of testing and related terminology [Bei90:c1; Jor02:c2; Lyu96:c2s2.2] (IEEE610.12-90)

A comprehensive introduction to the Software Testing KA is provided in the recommended references.

1.1.2. Faults vs. Failures

[Jor02:c2; Lyu96:c2s2.2; Per95:c1; Pfl01:c8] (IEEE610.12-90; IEEE982.1-88)

Many terms are used in the software engineering literature to describe a malfunction, notably *fault*, *failure*, *error*, and

several others. This terminology is precisely defined in IEEE Standard 610.12-1990, Standard Glossary of Software Engineering Terminology (IEEE610-90), and is also discussed in the Software Quality KA. It is essential to clearly distinguish between the *cause* of a malfunction, for which the term *fault* or *defect* will be used here, and an undesired effect observed in the system’s delivered service, which will be called a *failure*. Testing can reveal failures, but it is the faults that can and must be removed.

However, it should be recognized that the cause of a failure cannot always be unequivocally identified. No theoretical criteria exist to definitively determine what fault caused the observed failure. It might be said that it was the fault that had to be modified to remove the problem, but other modifications could have worked just as well. To avoid

ambiguity, some authors prefer to speak of *failure-causing inputs* (Fra98) instead of faults—that is, those sets of inputs that cause a failure to appear.

## 1.2. Key issues

### 1.2.1. Test selection criteria/Test adequacy criteria (or stopping rules)

[Pfl01:c8s7.3; Zhu97:s1.1] (Wey83; Wey91; Zhu97)

A test selection criterion is a means of deciding what a suitable set of test cases should be. A selection criterion can be used for selecting the test cases or for checking whether a selected test suite is adequate—that is, to decide whether the testing can be stopped. See also the sub-topic *Termination*, under topic 5.1 *Practical considerations*.

### 1.2.2. Testing effectiveness/Objectives for testing

[Bei90:c1s1.4; Per95:c21] (Fra98)

Testing is the observation of a sample of program executions. Sample selection can be guided by different objectives: it is only in light of the objective pursued that the effectiveness of the test set can be evaluated.

### 1.2.3. Testing for defect identification

[Bei90:c1; Kan99:c1]

In testing for defect identification, a successful test is one which causes the system to fail. This is quite different from testing to demonstrate that the software meets its specifications or other desired properties, in which case testing is successful if no (significant) failures are observed.

### 1.2.4. The oracle problem

[Bei90:c1] (Ber96, Wey83)

An oracle is any (human or mechanical) agent which decides whether a program behaved correctly in a given test, and accordingly produces a verdict of “pass” or “fail.” There exist many different kinds of oracles, and oracle automation can be very difficult and expensive.

### 1.2.5. Theoretical and practical limitations of testing

[Kan99:c2] (How76)

Testing theory warns against ascribing an unjustified level of confidence to a series of passed tests. Unfortunately, most established results of testing theory are negative ones, in that they state what testing can never achieve as opposed to what it actually achieved. The most famous quotation in this regard is the Dijkstra aphorism that “program testing can be used to show the presence of bugs, but never to show their absence.” The obvious reason is that complete testing is not feasible in real software. Because of this, testing must be driven based on risk and can be seen as a risk management strategy.

### 1.2.6. The problem of infeasible paths

[Bei90:c3]

Infeasible paths, the control flow paths that cannot be exercised by any input data, are a significant problem in path-oriented testing, and particularly in the automated derivation of test inputs for code-based testing techniques.

### 1.2.7. Testability

[Bei90:c3, c13] (Bac90; Ber96a; Voa95)

The term “software testability” has two related but different meanings: on the one hand, it refers to the degree to which it is easy for software to fulfill a given test coverage criterion, as in (Bac90); on the other hand, it is defined as the likelihood, possibly measured statistically, that the software will expose a failure under testing, *if it is faulty*, as in (Voa95, Ber96a). Both meanings are important.

### 1.3. Relationships of testing to other activities

Software testing is related to but different from static software quality management techniques, proofs of correctness, debugging, and programming. However, it is informative to consider testing from the point of view of software quality analysts and of certifiers.

- ♦ Testing vs. Static Software Quality Management techniques. See also the Software Quality KA, subarea 2. *Software Quality Management Processes*. [Bei90:c1; Per95:c17] (IEEE1008-87)
- ♦ Testing vs. Correctness Proofs and Formal Verification [Bei90:c1s5; Pfl01:c8].
- ♦ Testing vs. Debugging. See also the Software Construction KA, topic 3.4 *Construction testing* [Bei90:c1s2.1] (IEEE1008-87).
- ♦ Testing vs. Programming. See also the Software Construction KA, topic 3.4 *Construction testing* [Bei90:c1s2.3].
- ♦ Testing and Certification (Wak99).

## 2. Test Levels

### 2.1. The target of the test

Software testing is usually performed at different *levels* along the development and maintenance processes. That is to say, the target of the test can vary: a single module, a group of such modules (related by purpose, use, behavior, or structure), or a whole system. [Bei90:c1; Jor02:c13; Pfl01:c8] Three big test stages can be conceptually distinguished, namely Unit, Integration, and System. No process model is implied, nor are any of those three stages assumed to have greater importance than the other two.

#### 2.1.1. Unit testing

[Bei90:c1; Per95:c17; Pfl01:c8s7.3] (IEEE1008-87)

Unit testing verifies the functioning in isolation of software pieces which are separately testable. Depending on the context, these could be the individual subprograms or a larger component made of tightly related units. A test unit is defined more precisely in the IEEE Standard for Software Unit Testing (IEEE1008-87), which also describes an integrated approach to systematic and documented unit testing. Typically, unit testing occurs with access to the code being tested and with the support of

debugging tools, and might involve the programmers who wrote the code.

### 2.1.2. Integration testing

[Jor02:c13, 14; Pfl01:c8s7.4]

Integration testing is the process of verifying the interaction between software components. Classical integration testing strategies, such as top-down or bottom-up, are used with traditional, hierarchically structured software.

Modern systematic integration strategies are rather architecture-driven, which implies integrating the software components or subsystems based on identified functional threads. Integration testing is a continuous activity, at each stage of which software engineers must abstract away lower-level perspectives and concentrate on the perspectives of the level they are integrating. Except for small, simple software, systematic, incremental integration testing strategies are usually preferred to putting all the components together at once, which is pictorially called “big bang” testing.

### 2.1.3. System testing

[Jor02:c15; Pfl01:c9]

System testing is concerned with the behavior of a whole system. The majority of functional failures should already have been identified during unit and integration testing. System testing is usually considered appropriate for comparing the system to the non-functional system requirements, such as security, speed, accuracy, and reliability. External interfaces to other applications, utilities, hardware devices, or the operating environment are also evaluated at this level. See the Software Requirements KA for more information on functional and non-functional requirements.

## 2.2. Objectives of Testing

[Per95:c8; Pfl01:c9s8.3]

Testing is conducted in view of a specific objective, which is stated more or less explicitly, and with varying degrees of precision. Stating the objective in precise, quantitative terms allows control to be established over the test process.

Testing can be aimed at verifying different properties. Test cases can be designed to check that the functional specifications are correctly implemented, which is variously referred to in the literature as *conformance* testing, *correctness* testing, or *functional* testing. However, several other nonfunctional properties may be tested as well, including performance, reliability, and usability, among many others.

Other important objectives for testing include (but are not limited to) reliability measurement, usability evaluation, and acceptance, for which different approaches would be taken. Note that the test objective varies with the test target; in general, different purposes being addressed at a different level of testing.

References recommended above for this topic describe the set of potential test objectives. The sub-topics listed below are those most often cited in the literature. Note that some kinds of testing are more appropriate for custom-made software packages, *installation* testing, for example; and others for generic products, like *beta* testing.

### 2.2.1. Acceptance/qualification testing

[Per95:c10; Pfl01:c9s8.5] (IEEE12207.0-96:s5.3.9)

Acceptance testing checks the system behavior against the customer’s requirements, however these may have been expressed; the customers undertake, or specify, typical tasks to check that their requirements have been met or that the organization has identified these for the target market for the software. This testing activity may or may not involve the developers of the system.

### 2.2.2. Installation testing

[Per95:c9; Pfl01:c9s8.6]

Usually after completion of software and acceptance testing, the software can be verified upon installation in the target environment. Installation testing can be viewed as system testing conducted once again according to hardware configuration requirements. Installation procedures may also be verified.

### 2.2.3. Alpha and beta testing

[Kan99:c13]

Before the software is released, it is sometimes given to a small, representative set of potential users for trial use, either in-house (*alpha* testing) or external (*beta* testing). These users report problems with the product. Alpha and beta use is often uncontrolled, and is not always referred to in a test plan.

### 2.2.4. Conformance testing/Functional testing/Correctness testing

[Kan99:c7; Per95:c8] (Wak99)

Conformance testing is aimed at validating whether or not the observed behavior of the tested software conforms to its specifications.

### 2.2.5. Reliability achievement and evaluation

[Lyu96:c7; Pfl01:c9s.8.4] (Pos96)

In helping to identify faults, testing is a means to improve reliability. By contrast, by randomly generating test cases according to the operational profile, statistical measures of reliability can be derived. Using reliability growth models, both objectives can be pursued together (see also sub-topic 4.1.4 *Life test, reliability evaluation*).

### 2.2.6. Regression testing

[Kan99:c7; Per95:c11, c12; Pfl01:c9s8.1] (Rot96)

According to (IEEE610.12-90), regression testing is the “selective retesting of a system or component to verify that modifications have not caused unintended effects...” In practice, the idea is to show that software which previously

passed the tests still does. Beizer (Bei90) defines it as any repetition of tests intended to show that the software's behavior is unchanged, except insofar as required. Obviously a trade-off must be made between the assurance given by regression testing every time a change is made and the resources required to do that.

Regression testing can be conducted at each of the test levels described in topic 2.1 *The target of the test* and may apply to functional and nonfunctional testing.

#### 2.2.7. Performance testing

[Per95:c17; Pfl01:c9s8.3] (Wak99)

This is specifically aimed at verifying that the software meets the specified performance requirements, for instance, capacity and response time. A specific kind of performance testing is volume testing (Per95:p185, p487; Pfl01:p401), in which internal program or system limitations are tried.

#### 2.2.8. Stress testing

[Per95:c17; Pfl01:c9s8.3]

Stress testing exercises software at the maximum design load, as well as beyond it.

#### 2.2.9. Back-to-back testing

A single test set is performed on two implemented versions of a software product, and the results are compared.

#### 2.2.10. Recovery testing [Per95:c17; Pfl01:c9s8.3]

Recovery testing is aimed at verifying software restart capabilities after a "disaster."

#### 2.2.11. Configuration testing

[Kan99:c8; Pfl01:c9s8.3]

In cases where software is built to serve different users, configuration testing analyzes the software under the various specified configurations.

#### 2.2.12. Usability testing

[Per95:c8; Pfl01:c9s8.3]

This process evaluates how easy it is for end-users to use and learn the software, including user documentation; how effectively the software functions in supporting user tasks; and, finally, its ability to recover from user errors.

#### 2.2.13. Test-driven development

[Bec02]

Test-driven development is not a test technique per se, promoting the use of tests as a surrogate for a requirements specification document rather than as an independent check that the software has correctly implemented the requirements.

### 3. Test Techniques

One of the aims of testing is to reveal as much potential for failure as possible, and many techniques have been developed to do this, which attempt to "break" the program, by running one or more tests drawn from identified classes

of executions deemed equivalent. The leading principle underlying such techniques is to be as systematic as possible in identifying a representative set of program behaviors; for instance, considering subclasses of the input domain, scenarios, states, and dataflow.

It is difficult to find a homogeneous basis for classifying all techniques, and the one used here must be seen as a compromise. The classification is based on how tests are generated from the software engineer's intuition and experience, the specifications, the code structure, the (real or artificial) faults to be discovered, the field usage, or, finally, the nature of the application. Sometimes these techniques are classified as *white-box*, also called *glass-box*, if the tests rely on information about how the software has been designed or coded, or as *black-box* if the test cases rely only on the input/output behavior. One last category deals with combined use of two or more techniques. Obviously, these techniques are not used equally often by all practitioners. Included in the list are those that a software engineer should know.

#### 3.1. *Based on the software engineer's intuition and experience*

##### 3.1.1. Ad hoc testing

[Kan99:c1]

Perhaps the most widely practiced technique remains ad hoc testing: tests are derived relying on the software engineer's skill, intuition, and experience with similar programs. Ad hoc testing might be useful for identifying special tests, those not easily captured by formalized techniques.

##### 3.1.2. Exploratory testing

Exploratory testing is defined as simultaneous learning, test design, and test execution; that is, the tests are not defined in advance in an established test plan, but are dynamically designed, executed, and modified. The effectiveness of exploratory testing relies on the software engineer's knowledge, which can be derived from various sources: observed product behavior during testing, familiarity with the application, the platform, the failure process, the type of possible faults and failures, the risk associated with a particular product, and so on. [Kan01:c3]

#### 3.2. *Specification-based techniques*

##### 3.2.1. Equivalence partitioning

[Jor02:c7; Kan99:c7]

The input domain is subdivided into a collection of subsets, or equivalent classes, which are deemed equivalent according to a specified relation, and a representative set of tests (sometimes only one) is taken from each class.

##### 3.2.2. Boundary-value analysis

[Jor02:c6; Kan99:c7]

Test cases are chosen on and near the boundaries of the input domain of variables, with the underlying rationale that many faults tend to concentrate near the extreme values

of inputs. An extension of this technique is *robustness testing*, wherein test cases are also chosen outside the input domain of variables, to test program robustness to unexpected or erroneous inputs.

### 3.2.3. Decision table

[Bei90:c10s3] (Jor02)

Decision tables represent logical relationships between conditions (roughly, inputs) and actions (roughly, outputs). Test cases are systematically derived by considering every possible combination of conditions and actions. A related technique is *cause-effect graphing*. [Pfl01:c9]

### 3.2.4. Finite-state machine-based

[Bei90:c11; Jor02:c8]

By modeling a program as a finite state machine, tests can be selected in order to cover states and transitions on it.

### 3.2.5. Testing from formal specifications

[Zhu97:s2.2] (Ber91; Dic93; Hor95)

Giving the specifications in a formal language allows for automatic derivation of functional test cases, and, at the same time, provides a reference output, an oracle, for checking test results. Methods exist for deriving test cases from model-based (Dic93, Hor95) or algebraic specifications. (Ber91)

### 3.2.6. Random testing

[Bei90:c13; Kan99:c7]

Tests are generated purely at random, not to be confused with statistical testing from the operational profile as described in sub-topic 3.5.1 *Operational profile*. This form of testing falls under the heading of the specification-based entry, since at least the input domain must be known, to be able to pick random points within it.

## 3.3. Code-based techniques

### 3.3.1. Control-flow-based criteria

[Bei90:c3; Jor02:c10] (Zhu97)

Control-flow-based coverage criteria is aimed at covering all the statements or blocks of statements in a program, or specified combinations of them. Several coverage criteria have been proposed, like condition/decision coverage. The strongest of the control-flow-based criteria is path testing, which aims to execute all entry-to-exit control flow paths in the flowgraph. Since path testing is generally not feasible because of loops, other less stringent criteria tend to be used in practice, such as statement testing, branch testing, and condition/decision testing. The adequacy of such tests is measured in percentages; for example, when all branches have been executed at least once by the tests, 100% branch coverage is said to have been achieved.

### 3.3.2. Data flow-based criteria

[Bei90:c5] (Jor02; Zhu97)

In data-flow-based testing, the control flowgraph is annotated with information about how the program

variables are defined, used, and killed (undefined). The strongest criterion, all definition-use paths, requires that, for each variable, every control flow path segment from a definition of that variable to a use of that definition is executed. In order to reduce the number of paths required, weaker strategies such as all-definitions and all-uses are employed.

### 3.3.3. Reference models for code-based testing (flowgraph, call graph)

[Bei90:c3; Jor02:c5].

Although not a technique in itself, the control structure of a program is graphically represented using a flowgraph in code-based testing techniques. A flowgraph is a directed graph the nodes and arcs of which correspond to program elements. For instance, nodes may represent statements or uninterrupted sequences of statements, and arcs the transfer of control between nodes.

## 3.4. Fault-based techniques

(Mor90)

With different degrees of formalization, fault-based testing techniques devise test cases specifically aimed at revealing categories of likely or predefined faults.

### 3.4.1. Error guessing

[Kan99:c7]

In error guessing, test cases are specifically designed by software engineers trying to figure out the most plausible faults in a given program. A good source of information is the history of faults discovered in earlier projects, as well as the software engineer's expertise.

### 3.4.2. Mutation testing

[Per95:c17; Zhu97:s3.2-s3.3]

A mutant is a slightly modified version of the program under test, differing from it by a small, syntactic change. Every test case exercises both the original and all generated mutants: if a test case is successful in identifying the difference between the program and a mutant, the latter is said to be "killed." Originally conceived as a technique to evaluate a test set (see 4.2), mutation testing is also a testing criterion in itself: either tests are randomly generated until enough mutants have been killed, or tests are specifically designed to kill surviving mutants. In the latter case, mutation testing can also be categorized as a code-based technique. The underlying assumption of mutation testing, the coupling effect, is that by looking for simple syntactic faults, more complex but real faults will be found. For the technique to be effective, a large number of mutants must be automatically derived in a systematic way.

## 3.5. Usage-based techniques

### 3.5.1. Operational profile

[Jor02:c15; Lyu96:c5; Pfl01:c9]

In testing for reliability evaluation, the test environment must reproduce the operational environment of the software

as closely as possible. The idea is to infer, from the observed test results, the future reliability of the software when in actual use. To do this, inputs are assigned a probability distribution, or profile, according to their occurrence in actual operation.

### 3.5.2. Software Reliability Engineered Testing

[Lyu96:c6]

Software Reliability Engineered Testing (SRET) is a testing method encompassing the whole development process, whereby testing is “designed and guided by reliability objectives and expected relative usage and criticality of different functions in the field.”

### 3.6. Techniques based on the nature of the application

The above techniques apply to all types of software. However, for some kinds of applications, some additional know-how is required for test derivation. A list of a few specialized testing fields is provided here, based on the nature of the application under test:

- ♦ Object-oriented testing [Jor02:c17; Pfl01:c8s7.5] (Bin00)
- ♦ Component-based testing
- ♦ Web-based testing
- ♦ GUI testing [Jor20]
- ♦ Testing of concurrent programs (Car91)
- ♦ Protocol conformance testing (Pos96; Boc94)
- ♦ Testing of real-time systems (Sch94)
- ♦ Testing of safety-critical systems (IEEE1228-94)

### 3.7. Selecting and combining techniques

#### 3.7.1. Functional and structural

[Bei90:c1s.2.2; Jor02:c2, c9, c12; Per95:c17] (Pos96)

Specification-based and code-based test techniques are often contrasted as functional vs. structural testing. These two approaches to test selection are not to be seen as alternative but rather as complementary; in fact, they use different sources of information and have proved to highlight different kinds of problems. They could be used in combination, depending on budgetary considerations.

#### 3.7.2. Deterministic vs. random

(Ham92; Lyu96:p541-547)

Test cases can be selected in a deterministic way, according to one of the various techniques listed, or randomly drawn from some distribution of inputs, such as is usually done in reliability testing. Several analytical and empirical comparisons have been conducted to analyze the conditions that make one approach more effective than the other.

## 4. Test-related measures

Sometimes, test techniques are confused with test objectives. Test techniques are to be viewed as aids which

help to ensure the achievement of test objectives. For instance, branch coverage is a popular test technique. Achieving a specified branch coverage measure should not be considered the objective of testing per se: it is a means to improve the chances of finding failures by systematically exercising every program branch out of a decision point. To avoid such misunderstandings, a clear distinction should be made between test-related measures, which provide an evaluation of the program under test based on the observed test outputs, and those which evaluate the thoroughness of the test set. Additional information on measurement programs is provided in the Software Engineering Management KA, subarea 6, *Software engineering measurement*. Additional information on measures can be found in the Software Engineering Process KA, subarea 4, *Process and product measurement*.

Measurement is usually considered instrumental to quality analysis. Measurement may also be used to optimize the planning and execution of the tests. Test management can use several process measures to monitor progress. Measures relative to the test process for management purposes are considered in topic 5.1 *Practical considerations*.

#### 4.1. Evaluation of the program under test (IEEE982.1-98)

##### 4.1.1. Program measurements to aid in planning and designing testing

[Bei90:c7s4.2; Jor02:c9] (Ber96; IEEE982.1-88)

Measures based on program size (for example, source lines of code or function points) or on program structure (like complexity) are used to guide testing. Structural measures can also include measurements among program modules in terms of the frequency with which modules call each other.

##### 4.1.2. Fault types, classification, and statistics

[Bei90:c2; Jor02:c2; Pfl01:c8]

(Bei90; IEEE1044-93; Kan99; Lyu96)

The testing literature is rich in classifications and taxonomies of faults. To make testing more effective, it is important to know which types of faults could be found in the software under test, and the relative frequency with which these faults have occurred in the past. This information can be very useful in making quality predictions, as well as for process improvement. More information can be found in the Software Quality KA, topic 3.2 *Defect characterization*. An IEEE standard exists on how to classify software “anomalies” (IEEE1044-93).

##### 4.1.3. Fault density

[Per95:c20] (IEEE982.1-88; Lyu96:c9)

A program under test can be assessed by counting and classifying the discovered faults by their types. For each fault class, fault density is measured as the ratio between the number of faults found and the size of the program.

#### 4.1.4. Life test, reliability evaluation

[Pfl01:c9] (Pos96:p146-154)

A statistical estimate of software reliability, which can be obtained by reliability achievement and evaluation (see sub-topic 2.2.5), can be used to evaluate a product and decide whether or not testing can be stopped.

#### 4.1.5. Reliability growth models

[Lyu96:c7; Pfl01:c9] (Lyu96:c3, c4)

Reliability growth models provide a prediction of reliability based on the failures observed under reliability achievement and evaluation (see sub-topic 2.2.5). They assume, in general, that the faults that caused the observed failures have been fixed (although some models also accept imperfect fixes), and thus, on average, the product's reliability exhibits an increasing trend. There now exist dozens of published models. Many are laid down on some common assumptions, while others differ. Notably, these models are divided into *failure-count* and *time-between-failure* models.

### 4.2. Evaluation of the tests performed

#### 4.2.1. Coverage/thoroughness measures

[Jor02:c9; Pfl01:c8] (IEEE982.1-88)

Several test adequacy criteria require that the test cases systematically exercise a set of elements identified in the program or in the specifications (see subarea 3). To evaluate the thoroughness of the executed tests, testers can monitor the elements covered, so that they can dynamically measure the ratio between covered elements and their total number. For example, it is possible to measure the percentage of covered branches in the program flowgraph, or that of the functional requirements exercised among those listed in the specifications document. Code-based adequacy criteria require appropriate instrumentation of the program under test.

#### 4.2.2. Fault seeding

[Pfl01:c8] (Zhu97:s3.1)

Some faults are artificially introduced into the program before test. When the tests are executed, some of these seeded faults will be revealed, and possibly some faults which were already there will be as well. In theory, depending on which of the artificial faults are discovered, and how many, testing effectiveness can be evaluated, and the remaining number of genuine faults can be estimated. In practice, statisticians question the distribution and representativeness of seeded faults relative to genuine faults and the small sample size on which any extrapolations are based. Some also argue that this technique should be used with great care, since inserting faults into software involves the obvious risk of leaving them there.

#### 4.2.3. Mutation score

[Zhu97:s3.2-s3.3]

In mutation testing (see sub-topic 3.4.2), the ratio of killed mutants to the total number of generated mutants can be a measure of the effectiveness of the executed test set.

#### 4.2.4. Comparison and relative effectiveness of different techniques

[Jor02:c9, c12; Per95:c17; Zhu97:s5] (Fra93; Fra98; Pos96: p64-72)

Several studies have been conducted to compare the relative effectiveness of different test techniques. It is important to be precise as to the property against which the techniques are being assessed; what, for instance, is the exact meaning given to the term "effectiveness"? Possible interpretations are: the number of tests needed to find the first failure, the ratio of the number of faults found through testing to all the faults found during and after testing, or how much reliability was improved. Analytical and empirical comparisons between different techniques have been conducted according to each of the notions of effectiveness specified above.

## 5. Test Process

Testing concepts, strategies, techniques, and measures need to be integrated into a defined and controlled process which is run by people. The test process supports testing activities and provides guidance to testing teams, from test planning to test output evaluation, in such a way as to provide justified assurance that the test objectives will be met cost-effectively.

### 5.1. Practical considerations

#### 5.1.1. Attitudes/Egoless programming

[Bei90:c13s3.2; Pfl01:c8]

A very important component of successful testing is a collaborative attitude towards testing and quality assurance activities. Managers have a key role in fostering a generally favorable reception towards failure discovery during development and maintenance; for instance, by preventing a mindset of code ownership among programmers, so that they will not feel responsible for failures revealed by their code.

#### 5.1.2. Test guides

[Kan01]

The testing phases could be guided by various aims, for example: in risk-based testing, which uses the product risks to prioritize and focus the test strategy; or in scenario-based testing, in which test cases are defined based on specified software scenarios.

### 5.1.3. Test process management

[Bec02: III; Per95:c1-c4; Pfl01:c9] (IEEE1074-97; IEEE12207.0-96:s5.3.9, s5.4.2, s6.4, s6.5)

Test activities conducted at different levels (see subarea 2. *Test levels*) must be organized, together with people, tools, policies, and measurements, into a well-defined process which is an integral part of the life cycle. In IEEE/EIA Standard 12207.0, testing is not described as a stand-alone process, but principles for testing activities are included along with both the five primary life cycle processes and the supporting process. In IEEE Std 1074, testing is grouped with other evaluation activities as integral to the entire life cycle.

### 5.1.4. Test documentation and work products [Bei90:c13s5; Kan99:c12; Per95:c19; Pfl01:c9s8.8] (IEEE829-98)

Documentation is an integral part of the formalization of the test process. The IEEE Standard for Software Test Documentation (IEEE829-98) provides a good description of test documents and of their relationship with one another and with the testing process. Test documents may include, among others, Test Plan, Test Design Specification, Test Procedure Specification, Test Case Specification, Test Log, and Test Incident or Problem Report. The software under test is documented as the Test Item. Test documentation should be produced and continually updated, to the same level of quality as other types of documentation in software engineering.

### 5.1.5. Internal vs. independent test team

[Bei90:c13s2.2-c13s2.3; Kan99:c15; Per95:c4; Pfl01:c9]

Formalization of the test process may involve formalizing the test team organization as well. The test team can be composed of internal members (that is, on the project team, involved or not in software construction), of external members, in the hope of bringing in an unbiased, independent perspective, or, finally, of both internal and external members. Considerations of costs, schedule, maturity levels of the involved organizations, and criticality of the application may determine the decision.

### 5.1.6. Cost/effort estimation and other process measures [Per95:c4, c21] (Per95: Appendix B; Pos96:p139-145; IEEE982.1-88)

Several measures related to the resources spent on testing, as well as to the relative fault-finding effectiveness of the various test phases, are used by managers to control and improve the test process. These test measures may cover such aspects as number of test cases specified, number of test cases executed, number of test cases passed, and number of test cases failed, among others.

Evaluation of test phase reports can be combined with root-cause analysis to evaluate test process effectiveness in finding faults as early as possible. Such an evaluation could

be associated with the analysis of risks. Moreover, the resources that are worth spending on testing should be commensurate with the use/criticality of the application: different techniques have different costs and yield different levels of confidence in product reliability.

### 5.1.7. Termination

[Bei90:c2s2.4; Per95:c2]

A decision must be made as to how much testing is enough and when a test stage can be terminated. Thoroughness measures, such as achieved code coverage or functional completeness, as well as estimates of fault density or of operational reliability, provide useful support, but are not sufficient in themselves. The decision also involves considerations about the costs and risks incurred by the potential for remaining failures, as opposed to the costs implied by continuing to test. See also sub-topic 1.2.1 *Test selection criteria/Test adequacy criteria*.

### 5.1.8. Test reuse and test patterns

[Bei90:c13s5]

To carry out testing or maintenance in an organized and cost-effective way, the means used to test each part of the software should be reused systematically. This repository of test materials must be under the control of software configuration management, so that changes to software requirements or design can be reflected in changes to the scope of the tests conducted.

The test solutions adopted for testing some application types under certain circumstances, with the motivations behind the decisions taken, form a test pattern which can itself be documented for later reuse in similar projects.

## 5.2. Test Activities

Under this topic, a brief overview of test activities is given; as often implied by the following description, successful management of test activities strongly depends on the Software Configuration Management process.

### 5.2.1. Planning

[Kan99:c12; Per95:c19; Pfl01:c8s7.6] (IEEE829-98:s4; IEEE1008-87:s1-s3)

Like any other aspect of project management, testing activities must be planned. Key aspects of test planning include coordination of personnel, management of available test facilities and equipment (which may include magnetic media, test plans and procedures), and planning for possible undesirable outcomes. If more than one baseline of the software is being maintained, then a major planning consideration is the time and effort needed to ensure that the test environment is set to the proper configuration.

### 5.2.2. Test-case generation

[Kan99:c7] (Pos96:c2; IEEE1008-87:s4, s5)

Generation of test cases is based on the level of testing to be performed and the particular testing techniques. Test

cases should be under the control of software configuration management and include the expected results for each test.

#### 5.2.3. Test environment development

[Kan99:c11]

The environment used for testing should be compatible with the software engineering tools. It should facilitate development and control of test cases, as well as logging and recovery of expected results, scripts, and other testing materials.

#### 5.2.4. Execution

[Bei90:c13; Kan99:c11] (IEEE1008-87:s6, s7)

Execution of tests should embody a basic principle of scientific experimentation: everything done during testing should be performed and documented clearly enough that another person could replicate the results. Hence, testing should be performed in accordance with documented procedures using a clearly defined version of the software under test.

#### 5.2.5. Test results evaluation

[Per95:c20,c21] (Pos96:p18-20, p131-138)

The results of testing must be evaluated to determine whether or not the test has been successful. In most cases, “successful” means that the software performed as expected and did not have any major unexpected outcomes. Not all unexpected outcomes are necessarily faults, however, but could be judged to be simply noise. Before a failure can be removed, an analysis and debugging effort is needed to isolate, identify, and describe it. When test results are

particularly important, a formal review board may be convened to evaluate them.

#### 5.2.6. Problem reporting/Test log

[Kan99:c5; Per95:c20] (IEEE829-98:s9-s10)

Testing activities can be entered into a test log to identify when a test was conducted, who performed the test, what software configuration was the basis for testing, and other relevant identification information. Unexpected or incorrect test results can be recorded in a problem-reporting system, the data of which form the basis for later debugging and for fixing the problems that were observed as failures during testing. Also, anomalies not classified as faults could be documented in case they later turn out to be more serious than first thought. Test reports are also an input to the change management request process (see the Software Configuration Management KA, subarea 3, *Software configuration control*).

#### 5.2.7. Defect tracking

[Kan99:c6]

Failures observed during testing are most often due to faults or defects in the software. Such defects can be analyzed to determine when they were introduced into the software, what kind of error caused them to be created (poorly defined requirements, incorrect variable declaration, memory leak, programming syntax error, for example), and when they could have been first observed in the software. Defect-tracking information is used to determine what aspects of software engineering need improvement and how effective previous analyses and testing have been.

**MATRIX OF TOPICS VS. REFERENCE MATERIAL**

	[Bec02]	[Bei90]	[Jor02]	[Kan99]	[Kan01]	[Lyu96]	[Per95]	[Pfl01]	[Zhu97]
<b>1. Software Testing Fundamentals</b>									
<i>1.1 Testing-Related Terminology</i>									
Definitions of testing and related terminology		c1	c2			c2s2.2			
Faults vs. failures			c2			c2s2.2	c1	c8	
<i>1.2 Key Issues</i>									
Test selection criteria / test adequacy criteria (or stopping rules)								c8s7.3	s1.1
Testing effectiveness/objectives for testing		c1s1.4					c21		
Testing for defect identification		c1		c1					
The oracle problem		c1							
Theoretical and practical limitations of testing				c2					
The problem of infeasible paths		c3							
Testability		c3,c13							
<i>1.3 Relationships of Testing to other Activities</i>									
Testing vs. static analysis techniques		c1					c17		
Testing vs. correctness proofs and formal verification		c1s5						c8	
Testing vs. debugging		c1s2.1							
Testing vs. programming		c1s2.3							
Testing and certification									
<b>2. Test Levels</b>									
<i>2.1 The Target of the Tests</i>		c1	c13					c8	
Unit testing		c1					c17	c8s7.3	
Integration testing			c13,c14					c8s7.4	
System testing			c15					c9	
<i>2.2 Objectives of Testing</i>							c8	c9s8.3	
Acceptance/qualification testing							c10	c9s8.5	
Installation testing							c9	c9s8.6	
Alpha and beta testing				c13					
Conformance testing / Functional testing/ Correctness testing				c7			c8		
Reliability achievement and evaluation by testing						c7		c9s8.4	
Regression testing				c7			c11,c12	c9s8.1	
Performance testing							c17	c9s8.3	
Stress testing							c17	c9s8.3	
Back-to-back testing									
Recovery testing							c17	c9s8.3	
Configuration testing				c8				c9s8.3	
Usability testing							c8	c9s8.3	
Test-driven development	III								

	[Bec02]	[Bei90]	[Jor02]	[Kan99]	[Kan01]	[Lyu96]	[Per95]	[Pfl01]	[Zhu97]
<b>3. Test Techniques</b>									
<i>3.1 Based on tester's intuition and experience</i>									
Ad hoc testing				c1					
Exploratory testing					c3				
<i>3.2 Specification-based</i>									
Equivalence partitioning			c7	c7					
Boundary-value analysis			c6	c7					
Decision table		c10s3						c9	
Finite-state machine-based		c11	c8						
Testing from formal specifications									s2.2
Random testing		c13		c7					
<i>3.3 Code-based</i>									
Control-flow-based criteria		c3	c10					c8	
Data-flow-based criteria		c5							
Reference models for code-based testing		c3	c5						
<i>3.4 Fault-based</i>									
Error guessing				c7					
Mutation testing							c17		s3.2, s3.3
<i>3.5 Usage-based</i>									
Operational profile			c15			c5		c9	
Software Reliability Engineered Testing						c6			
<i>3.6 Based on Nature of Application</i>									
Object-oriented testing			c17					c8s7.5	
Component-based testing									
Web-based testing									
GUI testing			c20						
Testing of concurrent programs									
Protocol conformance testing									
Testing of distributed systems									
Testing of real-time systems									
<i>3.7 Selecting and Combining Techniques</i>									
Functional and structural		c1s2.2	c1,c11s11.3				c17		
Deterministic vs. Random									

	[Bec02]	[Bei90]	[Jor02]	[Kan99]	[Kan01]	[Lyu96]	[Per95]	[Pfl01]	[Zhu97]
<b>4. Test-Related Measures</b>									
<i>4.1 Evaluation of the Program under Test</i>									
Program measurements to aid in planning and designing testing.		c7s4.2	c9						
Types, classification and statistics of faults		c2	c1					c8	
Fault density							c20		
Life test, reliability evaluation								c9	
Reliability growth models						c7		c9	
<i>4.2 Evaluation of the Tests Performed</i>									
Coverage/thoroughness measures			c9					c8	
Fault seeding								c8	
Mutation score									s3.2, s3.3
Comparison and relative effectiveness of different techniques			c8,c11				c17		s5
<b>5. Test Process</b>									
<i>5.1 Practical Considerations</i>									
Attitudes/Egoless programming		c13s3.2						c8	
Test guides	III				C5				
Test process management							c1-c4	c9	
Test documentation and work products		c13s5		c12			c19	c9s8.8	
Internal vs. independent test team		c13s2.2, c1s2.3		c15			c4	c9	
Cost/effort estimation and other process measures							c4,c21		
Termination		c2s2.4					c2		
Test reuse and test patterns		c13s5							
<i>5.2 Test Activities</i>									
Planning				c12			c19	c87s7.6	
Test case generation				c7					
Test environment development				c11					
Execution		c13		c11					
Test results evaluation							c20,c21		
Problem reporting/Test log				c5			c20		
Defect tracking				c6					

#### RECOMMENDED REFERENCES FOR SOFTWARE TESTING

- [Bec02] K. Beck, *Test-Driven Development by Example*, Addison-Wesley, 2002.
- [Bei90] B. Beizer, *Software Testing Techniques*, International Thomson Press, 1990, Chap. 1-3, 5, 7s4, 10s3, 11, 13.
- [Jor02] P. C. Jorgensen, *Software Testing: A Craftsman's Approach*, second edition, CRC Press, 2004, Chap. 2, 5-10, 12-15, 17, 20.
- [Kan99] C. Kaner, J. Falk, and H.Q. Nguyen, *Testing Computer Software*, second ed., John Wiley & Sons, 1999, Chaps. 1, 2, 5-8, 11-13, 15.
- [Kan01] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing*, Wiley Computer Publishing, 2001.
- [Lyu96] M.R. Lyu, *Handbook of Software Reliability Engineering*, Mc-Graw-Hill/IEEE, 1996, Chap. 2s2.2, 5-7.
- [Per95] W. Perry, *Effective Methods for Software Testing*, John Wiley & Sons, 1995, Chap. 1-4, 9, 10-12, 17, 19-21.
- [Pfl01] S. L. Pfleeger, *Software Engineering: Theory and Practice*, second ed., Prentice Hall, 2001, Chap. 8, 9.
- [Zhu97] H. Zhu, P.A.V. Hall and J.H.R. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys*, vol. 29, iss. 4 (Sections 1, 2.2, 3.2, 3.3), Dec. 1997, pp. 366-427.

## APPENDIX A. LIST OF FURTHER READINGS

- (Bac90) R. Bache and M. Müllerburg, "Measures of Testability as a Basis for Quality Assurance," *Software Engineering Journal*, vol. 5, March 1990, pp. 86-92.
- (Bei90) B. Beizer, *Software Testing Techniques*, International Thomson Press, second ed., 1990.
- (Ber91) G. Bernot, M.C. Gaudel and B. Marre, "Software Testing Based On Formal Specifications: a Theory and a Tool," *Software Engineering Journal*, Nov. 1991, pp. 387-405.
- (Ber96) A. Bertolino and M. Marrè, "How Many Paths Are Needed for Branch Testing?" *Journal of Systems and Software*, vol. 35, iss. 2, 1996, pp. 95-106.
- (Ber96a) A. Bertolino and L. Strigini, "On the Use of Testability Measures for Dependability Assessment," *IEEE Transactions on Software Engineering*, vol. 22, iss. 2, Feb. 1996, pp. 97-108.
- (Bin00) R.V. Binder, *Testing Object-Oriented Systems Models, Patterns, and Tools*, Addison-Wesley, 2000.
- (Boc94) G.V. Bochmann and A. Petrenko, "Protocol Testing: Review of Methods and Relevance for Software Testing," presented at *ACM Proc. Int'l Symp. on Software Testing and Analysis (ISSTA '94)*, Seattle, Wash., 1994.
- (Car91) R.H. Carver and K.C. Tai, "Replay and Testing for Concurrent Programs," *IEEE Software*, March 1991, pp. 66-74.
- (Dic93) J. Dick and A. Faivre, "Automating the Generation and Sequencing of Test Cases from Model-Based Specifications," presented at *FME '93: Industrial-Strength Formal Methods*, LNCS 670, Springer-Verlag, 1993.
- (Fran93) P. Frankl and E. Weyuker, "A Formal Analysis of the Fault Detecting Ability of Testing Methods," *IEEE Transactions on Software Engineering*, vol. 19, iss. 3, March 1993, p. 202.
- (Fran98) P. Frankl, D. Hamlet, B. Littlewood, and L. Strigini, "Evaluating Testing Methods by Delivered Reliability," *IEEE Transactions on Software Engineering*, vol. 24, iss. 8, August 1998, pp. 586-601.
- (Ham92) D. Hamlet, "Are We Testing for True Reliability?" *IEEE Software*, July 1992, pp. 21-27.
- (Hor95) H. Horcher and J. Peleska, "Using Formal Specifications to Support Software Testing," *Software Quality Journal*, vol. 4, 1995, pp. 309-327.
- (How76) W. E. Howden, "Reliability of the Path Analysis Testing Strategy," *IEEE Transactions on Software Engineering*, vol. 2, iss. 3, Sept. 1976, pp. 208-215.
- (Jor02) P.C. Jorgensen, *Software Testing: A Craftsman's Approach*, second ed., CRC Press, 2004.
- (Kan99) C. Kaner, J. Falk, and H.Q. Nguyen, "Testing Computer Software," second ed., John Wiley & Sons, 1999.
- (Lyu96) M.R. Lyu, *Handbook of Software Reliability Engineering*, Mc-Graw-Hill/IEEE, 1996.
- (Mor90) L.J. Morell, "A Theory of Fault-Based Testing," *IEEE Transactions on Software Engineering*, vol. 16, iss. 8, August 1990, pp. 844-857.
- (Ost88) T.J. Ostrand and M.J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests," *Communications of the ACM*, vol. 31, iss. 3, June 1988, pp. 676-686.
- (Ost98) T. Ostrand, A. Anodide, H. Foster, and T. Goradia, "A Visual Test Development Environment for GUI Systems," presented at *ACM Proc. Int'l Symp. on Software Testing and Analysis (ISSTA '98)*, Clearwater Beach, Florida, 1998.
- (Per95) W. Perry, *Effective Methods for Software Testing*, John Wiley & Sons, 1995.
- (Pfl01) S.L. Pfleeger, *Software Engineering: Theory and Practice*, second ed., Prentice-Hall, 2001, Chap. 8, 9.
- (Pos96) R.M. Poston, *Automating Specification-Based Software Testing*, IEEE, 1996.
- (Rot96) G. Rothermel and M.J. Harrold, "Analyzing Regression Test Selection Techniques," *IEEE Transactions on Software Engineering*, vol. 22, iss. 8, Aug. 1996, p. 529.
- (Sch94) W. Schütz, "Fundamental Issues in Testing Distributed Real-Time Systems," *Real-Time Systems Journal*, vol. 7, iss. 2, Sept. 1994, pp. 129-157.
- (Voa95) J.M. Voas and K.W. Miller, "Software Testability: The New Verification," *IEEE Software*, May 1995, pp. 17-28.
- (Wak99) S. Wakid, D.R. Kuhn, and D.R. Wallace, "Toward Credible IT Testing and Certification," *IEEE Software*, July-Aug. 1999, pp. 39-47.
- (Wey82) E.J. Weyuker, "On Testing Non-testable Programs," *The Computer Journal*, vol. 25, iss. 4, 1982, pp. 465-470.
- (Wey83) E.J. Weyuker, "Assessing Test Data Adequacy through Program Inference," *ACM Trans. on Programming Languages and Systems*, vol. 5, iss. 4, October 1983, pp. 641-655.
- (Wey91) E.J. Weyuker, S.N. Weiss, and D. Hamlet, "Comparison of Program Test Strategies," presented at *Proc. Symp. on Testing, Analysis and Verification (TAV 4)*, Victoria, British Columbia, 1991.
- (Zhu97) H. Zhu, P.A.V. Hall, and J.H.R. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys*, vol. 29, iss. 4, Dec. 1997, pp. 366-427.

## APPENDIX B. LIST OF STANDARDS

(IEEE610.12-90) IEEE Std 610.12-1990 (R2002), *IEEE Standard Glossary of Software Engineering Terminology*, IEEE, 1990.

(IEEE829-98) IEEE Std 829-1998, *Standard for Software Test Documentation*, IEEE, 1998.

(IEEE982.1-88) IEEE Std 982.1-1988, *IEEE Standard Dictionary of Measures to Produce Reliable Software*, IEEE, 1988.

(IEEE1008-87) IEEE Std 1008-1987 (R2003), *IEEE Standard for Software Unit Testing*, IEEE, 1987.

(IEEE1044-93) IEEE Std 1044-1993 (R2002), *IEEE Standard for the Classification of Software Anomalies*, IEEE, 1993.

(IEEE1228-94) IEEE Std 1228-1994, *Standard for Software Safety Plans*, IEEE, 1994.

(IEEE12207.0-96) IEEE/EIA 12207.0-1996 // ISO/IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, IEEE, 1996.